

# ARSENAL:

## Automatic Requirements Specification Extraction from Natural Language

Shalini Ghosh<sup>1</sup>, Daniel Elenius<sup>1</sup>, Wenchao Li<sup>1</sup>,  
Patrick Lincoln<sup>1</sup>, Natarajan Shankar<sup>1</sup>, Wilfried Steiner<sup>2</sup>

<sup>1</sup>CSL, SRI International, Menlo Park. {shalini,elenius,li,lincoln,shankar}@csl.sri.com

<sup>2</sup>TTTech C. AG, Chip IP Design, A-1040 Vienna, Austria. wilfried.steiner@tttech.com

**Abstract.** Natural language (supplemented with diagrams and some mathematical notations) is convenient for succinct communication of technical descriptions between the various stakeholders (e.g., customers, designers, implementers) involved in the design of software systems. However, natural language descriptions can be informal, incomplete, imprecise and ambiguous, and cannot be processed easily by design and analysis tools. Formal languages, on the other hand, formulate design requirements in a precise and unambiguous mathematical notation, but are more difficult to master and use. We propose a methodology for connecting semi-formal requirements with formal descriptions through an intermediate representation. We have implemented this methodology in a research prototype called Automatic Requirements Specification Extraction from Natural Language (**ARSENAL**). The main novelty of ARSENAL lies in its ability to generate a fully-specified complete formal model automatically from natural language requirements. ARSENAL extracts relations from text using semantic parsing and progressively refines them over multiple stages to create a final composite model. Currently, ARSENAL generates formal models in linear-time temporal logic (**LTL**), but the approach can be adapted for other models, e.g., probabilistic relational models like Markov Logic Networks (**MLN**). The formal models of the requirements can be used to check important design and system properties, e.g., consistency, satisfiability, realizability. ARSENAL has a modular and flexible architecture that facilitates porting it from one domain to another. We evaluated ARSENAL on complex requirements from two real-world case studies: the Time-Triggered Ethernet (**TTEthernet**) communication platform used in space, and **FAA**-Isolette infant incubators used in **NICU**. We systematically evaluated various aspects of ARSENAL — the accuracy of the natural language processing stage, the degree of automation, and robustness to noise.

# Table of Contents

ARSENAL: .....	1
<i>Shalini Ghosh<sup>1</sup>, Daniel Elenius<sup>1</sup>, Wenchao Li<sup>1</sup>, Patrick Lincoln<sup>1</sup>, Natarajan Shankar<sup>1</sup>, Wilfried Steiner<sup>2</sup></i>	
1 Introduction .....	3
2 The ARSENAL Methodology .....	5
3 Natural Language Processing .....	6
3.1 Preprocessor .....	6
3.2 Stanford Typed Dependency Parser .....	7
3.3 Semantic Processor .....	8
Metadata tags .....	8
Type Rules .....	9
3.4 Dependency Graph $\rightarrow$ Predicate Graph $\rightarrow$ Logical Formula .....	9
4 Formal Analysis .....	11
4.1 Formula Generation .....	11
4.2 Consistency Analysis with Büchi Automata .....	12
4.3 The SAL Modeling Language .....	14
4.4 Verification with SAL .....	15
4.5 Synthesis with RATSY .....	18
5 Evaluation .....	18
5.1 Degree of Automation Metric .....	18
5.2 Degree of Perturbation Metric .....	19
5.3 NLP Stage Accuracy .....	20
Typed Levenshtein Distance .....	21
Max-weighted matching in bipartite graph .....	21
Evaluation on Test set .....	21
5.4 Case Study: FAA-Isolette .....	22
6 Related Work .....	24
6.1 Summary of Related Work .....	24
6.2 Details of Related Work .....	24
Requirements Engineering .....	24
Natural Language Processing (NLP) .....	26
Compliance checking and monitoring .....	27
7 Conclusion and Future Work .....	28
8 Acknowledgments .....	29
References .....	30

# 1 Introduction

Software systems operate in the real world, and often work in conjunction with complex physical systems. Many different stakeholders participate in the design and operation of these systems. Requirements specify important properties of (cyber-physical) software systems, e.g., conditions required to achieve an objective, or desired invariants of the system. Requirements in formal languages are precise – the formal models for requirements are useful for checking consistency and verifying properties, but are cumbersome to specify. As a result, stakeholders (e.g., customers, designers, engineers) often prefer writing requirements in natural language (NL). The NL requirements can be written easily without burden of formal rigor — there are standard guidelines for writing requirements (for example, IEEE recommended practice for software requirements specifications [iee94]), but in spite of that requirements written in NL can be imprecise, incomplete, and ambiguous.

So, natural language descriptions and formal modeling languages each offer distinct advantages to the system designer. The informality of natural language can kick-start discussion among stakeholders in early design, but can lead to confusion, lack of automation, and errors. The rigor of formal languages can eliminate broad classes of ambiguity, enable consistency checking, and facilitate automatic test case generation. However, mastery of formal notations requires a significant amount of training and mathematical sophistication.

Another important issue to consider is the cost of errors — most of the costly errors often enter at the requirements stage as a result of confusion among stakeholders [BP98]: “If a defect is found in the requirements phase, it may cost \$1 to fix. It is proffered that the same defect will cost \$10 if found in design, \$100 during coding, \$1000 during testing [bug10].” In order to catch as many errors as possible during the requirements phase, iterations between the stakeholders through clear communication in natural language must be supported. Formal models and descriptions that can detect errors, incompleteness, ambiguity, and inconsistency in the requirements should also be used. By bridging the gap between semi-formal requirements and formal specifications, we can dramatically reduce the number of costly uncaught errors in requirements and enable high levels of assurance for critical complex systems. Figure 1 summarizes the tradeoff between natural language and formal requirements specifications.

We aim to leverage the best of both natural and formal languages to aid the system designer in achieving high assurance for critical systems. This paper’s primary objective is to answer this question:

*Can we build a requirements engineering framework that combines the strengths of semi-formal natural language and precise formal notations?*

To that effect, we present the “Automatic Requirements Specification Extraction from Natural Language” (ARSENAL) methodology. ARSENAL uses state-of-the-art advances in natural language processing (NLP) and formal methods (FM) to connect natural language descriptions with their precise formal representations. ARSENAL provides a method for extracting relevant information from NL requirements documents and creating formal models with that information — it is an exploratory and experimental open-loop framework for extracting formal meaning from semi-formal text.

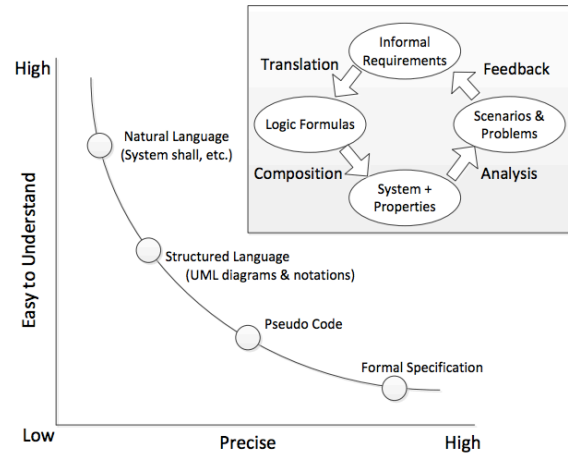
Let us consider the following sentence, which is part of the requirements specification for a regulator that regulates the temperature in an *isolette* (an incubator for an infant that provides controlled temperature, humidity, and oxygen):

**REQ1:** *If the Status attribute of the Lower Desired Temperature is Invalid, the Regulator Interface Failure shall be set to True.*

This requirements sentence is written with terminology that is particular to the domain, in a language that facilitates comprehensible communication between the relevant stakeholders involved in different stages of the *isolette* design. ARSENAL aims to convert collections of such requirements automatically to formal models, providing a natural language front-end to formal analysis that is robust and flexible across different forms of natural language expressions in different domains, and customized to stylized usages within each domain.

**Table 1.** Key innovations in ARSENAL.

	Challenges	Key Insights
1	Bridge the gap between semi-formal natural language requirements and precise formal models.	Create a rich/expressive intermediate representation (IR), useful for generating outputs in multiple formalisms.
2	Create a general-purpose architecture that can be ported to different domains.	Encapsulate domain-specific components in modules (e.g., NL preprocessor, output generators), keeping rest of system domain-independent and reusable.
3	Incorporate semantics into formal model generation.	Add semantics to the formal model via rewrite rules and type inference algorithms in the model generator stage.



**Fig. 1.** Tradeoff between natural language and formal specifications [Bab07], inset showing the design-iteration cycle of the ARSENAL methodology.

The input of ARSENAL consists of requirements in natural language with specific technical content. ARSENAL uses domain-specific semantic parsing to extract relations from text, and progressively refines them over multiple stages to create formulas in first-order logic (FOL) or linear-temporal logic (LTL). These are then used to create a composite model in the FM stage, which can be used by formal verification tools like theorem provers (e.g., PVS [ORR+96]) and model checking tools (e.g., SAL [BGL+00]) as well as LTL synthesis tools (e.g. RATS [BCG+10]) for automated analysis of the formal specifications. The results provide concrete empirical evidence that it is possible to bridge the gap between natural language requirements and formal specifications, achieving a promising level of performance and domain independence.

The main challenges and key insights of ARSENAL are outlined in Table 1. The organization of the rest of the paper is as follows: Section 2 gives an overview of ARSENAL, while Sections 3 and 4 respectively describe the NLP and FM stages in more detail. Section 5 discusses the results of our experiments with ARSENAL on the FAA-Isolette and TTEthernet requirements documents, while Section 6 discusses the novelty of ARSENAL compared to related research. Finally, Section 7 summarizes the contributions of this work and outlines possible future directions of research.

## 2 The ARSENAL Methodology

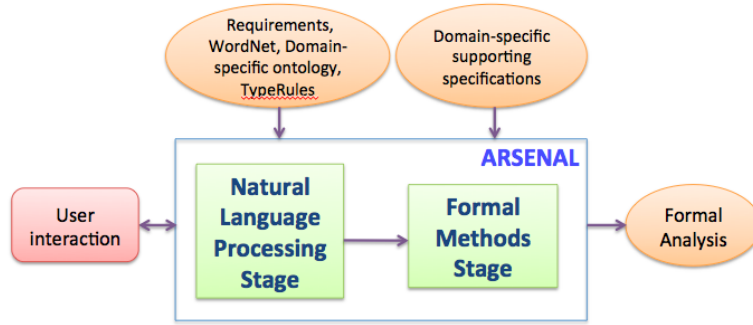


Fig. 2. ARSENAL pipeline.

In this section, we give an overview of the flow (shown in Figure 2) using the example requirement sentence REQ1 from the FAA-Isolette corpus, introduced in Section 1.

Given requirements written in natural language, ARSENAL first processes them using a natural language processing (NLP) stage. The NLP stage has a preprocessor that does some domain-independent processing (e.g., identifying arithmetic formulas) as well as domain-specific processing (e.g., identifying domain-specific nominal phrases corresponding to an entity). In REQ1, the preprocessor identifies terms like *Lower Desired Temperature*, *Status Attribute* and *Regulator Interface Failure* as phrases with special meanings in the FAA-Isolette domain, and converts each of these phrases to a single term (corresponding to an entity in this domain). This results in the following preprocessed requirements sentence:

*If the Status\_attribute of the Lower\_Desired\_Temperature is Invalid, the Regulator\_Interface\_Failure shall be set to True.*

The output of the preprocessor is analyzed by a semantic processor that first does shallow semantic parsing of the preprocessed requirements text using the Stanford Typed Dependency Parser (STDP) [dMMM06]. STDP outputs typed dependencies such as: `nsubj(equals, Status_attribute)`.

Each typed dependency indicates a semantic relation between parsed terms, e.g., the typed dependency above indicates that `Status_attribute` is a subject of the verb `equals`. The next stage of the semantic

processor converts these typed dependencies to entries in a symbol table in an intermediate representation (IR) of the form:

```
Status_Attribute-3: Status_Attribute | unique | of: [Lower_Desired_Temperature-6]
```

The IR table maps each symbol to its metadata and to its relationships with other symbols. In the example above, the IR table entry for `Status_Attribute` shows that it is unique, and is connected to another entity `Lower_Desired_Temperature` via the relation `of`. A detailed description of the IR table is given in Section 3.

The next part of the ARSENAL pipeline is the Formal Methods (FM) stage, which converts the IR table to a formal model (in the current ARSENAL prototype, we generate a SAL model). ARSENAL effectively converts multiple NL requirements sentences, which describe a system module, into a unified SAL model. Using this model, we can potentially generate a proof or counter-example for certain system properties of interest.

The SAL model generated for REQ1 is shown in Figure 3.

```
model_Example : CONTEXT =
BEGIN
  Type1 : TYPE = {Invalid};
  Type0 : TYPE = [# Status_attribute : Type1 #];
  main : MODULE =
  BEGIN
    LOCAL Regulator_Interface_Failure : BOOLEAN
    LOCAL Upper_Desired_Temperature : Type0
    LOCAL Lower_Desired_Temperature : Type0
    DEFINITION
      Regulator_Interface_Failure IN {Z : BOOLEAN |
        (Upper_Desired_Temperature.Status_attribute = Invalid OR
         Lower_Desired_Temperature.Status_attribute = Invalid)
        => Z = TRUE};
    END;
  END
```

**Fig. 3.** SAL model for REQ1.

In the rest of the paper, we describe how ARSENAL can automatically translate multiple natural language requirements sentences that describe a transition system into a unified SAL model with associated properties.

Note that ARSENAL is a general purpose methodology. We can plug in different modules to various parts of the workflow, e.g., any state-of-the-art typed dependency parser in the NLP stage or formal analysis tool in the FM stage. In this instance of the ARSENAL pipeline, we use STDP and SAL in the NLP stage and FM stage respectively (as described in the following sections), but other tools can also be plugged into these stages.

### 3 Natural Language Processing

The NLP stage takes requirements in natural language as input and generates the IR table as output. The different components of the NLP stage (shown in Figure 4) are described in detail in this section.

#### 3.1 Preprocessor

The first part of the NLP stage is a preprocessor. It seeks to extract better (more meaningful) parses to aid the Stanford parser using both domain-specific and domain-independent transformations on the

requirements sentence. An example domain-specific preprocessing task is identifying entity phrases like “Lower Desired Temperature” and converting them to the term `Lower_Desired_Temperature`. Domain-independent preprocessing tasks include identifying and transforming arithmetic expressions, so that NLP parsers like the Stanford parser can handle them better. For example, the preprocessor replaces the arithmetic expression “[ $x + 5$ ]” by `ARITH_x_PLUS_5`. The parser then treats this as a single term, instead of trying to parse the five symbols in the arithmetic expression. The preprocessor also encodes complex phrases like “is greater than or equal to” into simpler terms like `dominates`. In later processing (e.g., in the Formal Methods stage), ARSENAL decodes the original arithmetic expressions from the corresponding encoded strings. Apart from detecting n-grams and converting them to phrases (based on the input glossary), and doing arithmetic processing, the pre-processor also inserts missing punctuation (e.g., commas at phrase boundaries) and does other text transforms — these help the downstream parser get better results.

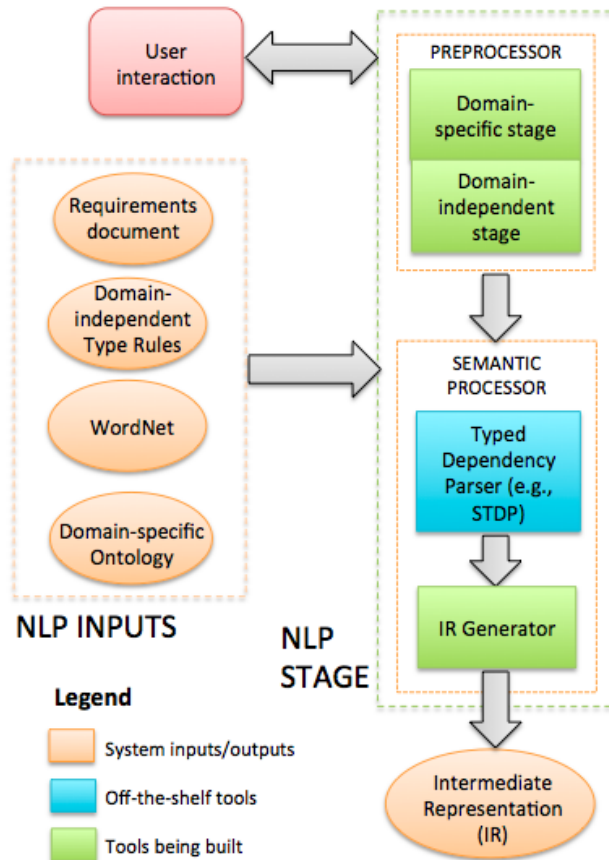


Fig. 4. NLP Stage of ARSENAL pipeline.

### 3.2 Stanford Typed Dependency Parser

The next part of the NLP stage is the application of the Stanford Typed Dependency Parser (STDP) to the preprocessed sentence. The syntactic parser in STDP parses the requirements text to get unique entities called mentions, while the dependency parser detects grammatical relations between the mentions. The final

output is a set of typed dependency (TD) triples between extracted terms, which encode the grammatical relationship between mentions extracted from a sentence. The Stanford typed dependencies representation provides a simple description of the grammatical relationships in a sentence, which can be understood easily and used effectively to extract textual relations without requiring deep linguistic expertise [dMMM06].

The TDs output by STDP are triples of the form: **relation (governor, dependent)**

Each triple indicates a relation of type “relation name” between the governor and dependent terms. For example, let us consider the TD:

`prep_of(Status_attribute-3, Lower_Desired_Temperature-6)`

It indicates that the mention `Status_attribute` is related to `Lower_Desired_Temperature` via a prepositional connective of type “of”. Note that the suffix of each mention is a number indicating the word position of the mention in the sentence, e.g., `equals-10` refers to the mention `equals` at position 10 in the sentence. The position index helps to uniquely identify the mention if it has multiple occurrences in the sentence. We will denote governor terms by ?g and dependent terms by ?d.

For the example requirement REQ1, the full set of TDs generated by STDP are shown in Figure 5.

```
mark(equals-10, If-1)
det(Status_attribute-3, the-2)
nsubj(equals-10, Status_attribute-3)
det(Lower_Desired_Temperature-6, the-5)
prep_of(Status_attribute-3, Lower_Desired_Temperature-6)
det(Upper_Desired_Temperature-9, the-8)
prep_of(Status_attribute-3, Upper_Desired_Temperature-9)
conj_or(Lower_Desired_Temperature-6, Upper_Desired_Temperature-9)
advcl(set-17, equals-10)
dobj(equals-10, Invalid-11)
det(Regulator_Interface_Failure-14, the-13)
nsubjpass(set-17, Regulator_Interface_Failure-14)
aux(set-17, shall-15)
auxpass(set-17, be-16)
root(ROOT-0, set-17)
prep_to(set-17, True-19)
```

**Fig. 5.** STDP output for REQ1.

### 3.3 Semantic Processor

One key challenge we face in ARSENAL is the mapping from natural language sentences to LTL formulas in a way that the LTL semantics are correctly preserved. The output of STDP is a set of grammatical relations that lack logical meaning — we developed a semantic processor that takes the output from STDP and systematically applies a set of type rules to the mentions and dependencies to associate meanings to them. Each type rule specifies a mapping from a set of dependencies (grammatical relations between mentions) to a set of predicates with built-in “semantics”. The semantic processor generates an Intermediate Representation (IR) by annotating the output of STDP with Metadata tags and by the consecutive application of type rules. The overall algorithm for generating the IR table is outlined in Figure 6, while metadata tags and type rules are described in the rest of this section.

**Metadata tags** Different types of metadata tags are used to annotate the IR table entries:

1. TermType: Whether term is of type entity, event, numeric, or predicate.
2. NegatedOrNot: Whether term is logically negated.



3. QuantifierType: Unique, all, exists.
4. Relations/Attributes: Temporal or normal.
5. Logical relations: Corresponds to the connectives and, or, implied-by.

These tags are used to associate semantics with the entries in the IR table. The metatag annotations are similar to role annotations in automatic semantic role labeling [CFR12]. In this stage, ARSENAL uses WordNet [Mil95] to identify word stems and to infer unique proper nouns. ARSENAL can also use special domain-specific ontologies or glossaries in this stage to annotate the IR entries with a richer set of metadata tags, which can be used in downstream processing.

**Type Rules** The majority of type rules are domain-independent semantic rules used by the semantic processor to create the IR table — each type rule specifies a mapping from a set of dependencies to a set of relational predicates, with built-in semantics. For example, `nsubjpass(V, N)` in the STDP output indicates that the noun phrase `N` is the syntactic subject of a passive clause with the root verb `V`. The type rule corresponding to the TD `nsubjpass(V, N)` indicates that `N` is a who/what argument of relation `V` in the output formula. Type rules have the form: `TD(arg1,arg2): ACTION(arg3,arg4)`. For5C example: `prep-upon(?g,?d): implies(?d,?g)`

**Matching of Typed Dependencies with Type Rules:** Matching this rule with the TD `prep-upon(entering-17, set-4)` produces a match with `?g = entering-17`, `?d = set-4`. The action to execute is then `implies(set-4, entering-17)`. The `implies(?x,?y)` action adds an entry `impliedBy:?x` to the IR entry for `?y`.

ARSENAL has different kinds of type rules, e.g., for handling implications, conjunctions/disjunctions, universal/existential quantifiers, temporal attributes/relations, relation arguments, and events.

**Rules with complex patterns:** Some type rules are complex and have multiple TDs or conditions that match on the left-hand side. For example: `nsubj(?g,?d) & event(?g): rel(agent,?g,?d)`

Here we have an additional check that whatever mention matches `?g` is marked as an event in the IR (in step 4 of the algorithm in Figure 6) — `rel(agent,?g,?d)` adds a relation `agent=?d` to the IR entry for `?g`. Figure 7 shows the IR table for REQ1 after application of the type rules. Currently the type rules are created by domain experts, aided by a statistics generator that finds the dominant dependencies in a corpus. Most type rules are domain-independent and can be re-used in other domains — the domain-dependent ones need to be customized by tuning based on training set from a corpus.

A plug-and-play architecture like ARSENAL gives certain desirable flexibilities. For example, if the shallow semantic NL parser generates multiple candidate parses of the requirements, ARSENAL can use semantic rules to select the best parse.

### 3.4 Dependency Graph $\rightarrow$ Predicate Graph $\rightarrow$ Logical Formula

The result of applying STDP can be represented as a dependency graph, which is a directed graph that shows the type dependencies (Figure 8). The IR table can be considered to be an annotated adjacency list for the dependency graph, where additional annotations are stored in the IR table along with the type dependency information — these annotations correspond to metadata generated by type rules, and are used in down-stream processing.

During generation of the logical formula, the dependency graph is transformed to a predicate graph. The predicate graph selects and refines the relations in the IR that are relevant for the output language. The predicate graph for REQ1 is shown in Figure 9. In the predicate graph, similar to the dependency graph, the edges represent binary predicates (with predefined meanings). The unique unary predicates are indicated by boxes in the figure. Additionally, mentions containing indicative words such as “equals” and “set” are associated with predefined predicates *equal* and *set*. For example, the predicate *set* means that its first argument (object) is a variable being set to a value which is its second argument (to). The semantic information

**Input:** Requirements text, WordNet, Domain-specific ontology, TypeRules, Pre-processing rules.

**Output:** Intermediate Representation (IR) table.

1. **Run the requirement text through Stanford Dependency Parser:**  
This produces a graph of Typed Dependencies (TDs), and part of speech (POS) tags for all the words in the sentence.
2. **Create a Mention Table:** Select each MentionId in Stanford Parser output and creating a hash from MentionId to all typed dependencies (TDs) it is involved in.
3. **Initialize IR:** Empty at beginning.
4. **Populate the IR:** Iterate over MentionIds in the Mention Table in sequence. For each MentionId:
  - (a) Get the POS tag for the MentionId.
  - (b) Set *word* to the stem of the word (from the WordNet stemmer) and add an IR entry for the word.
  - (c) If word is a math expression encoded by the math preprocessor, set its IR type to *arithmetic*.
  - (d) Else if word is marked as a unique entity in the ontology, set its IR type to entity and its quantifier to *unique*.
  - (e) Else if word is marked as predicate in the ontology, set its IR type to *pred*.
  - (f) Else if word is a number, set its IR type to *num*.
  - (g) Else if word has a noun POS tag, set its IR type to *entity*. In addition, if the word is not found in WordNet, set its quantifier to *unique* (as it is presumably a proper name).
  - (h) Else if word has a verb POS tag, set its type to *event*.
5. **Execute the type rules by doing a DFS traversal of the TD graph:**  
For each MentionId in the Mention Table, for each TD associated with MentionId in the Mention Table, for each type rule TR:
  - (a) Match the type rule with the TD, producing TD.
  - (b) If step 5(a) was successful, i.e., the left-hand-side of TR matches TD, execute the right-hand-side of TD.

**Fig. 6.** Detailed algorithmic flow of IR generation.

Status_attribute-3	: Status_attribute   entity   unique   of=Upper_Desired_Temperature-9
Lower_Desired_Temperature-6	: Lower_Desired_Temperature   entity   unique
Upper_Desired_Temperature-9	: Upper_Desired_Temperature   entity   unique   or: [Lower_Desired_Temperature-6]
equals-10	: equal   predicate   arg2=Invalid-11, arg1=Status_attribute-3
Invalid-11	: Invalid   entity
Regulator_Interface_Failure-14	: Regulator_Interface_Failure   entity   unique
be-16	: be   event
set-17	: set   event   to=True-19, object= Regulator_Interface_Failure-14   impliedBy: [equals-10]
True-19	: True   bool

**Fig. 7.** IR table for REQ1.

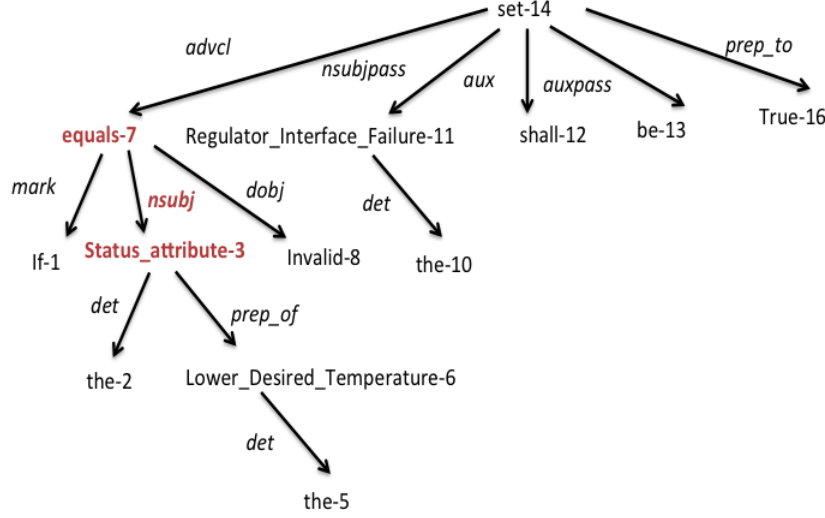


Fig. 8. Dependencies generated using STDP.

in the predicate graph is further interpreted in the FM stage, based on the target language and additional information about the model.

## 4 Formal Analysis

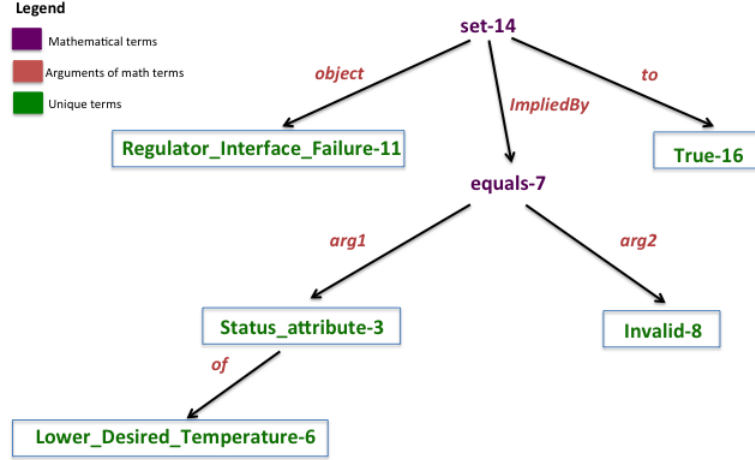
In this section, we discuss the different parts of the Formal Methods (FM) stage, as shown in Figure 10. The overall flow of the FM stage is shown in Figure 12 – this stage uses a combination of consistency, satisfiability and realizability checks to formally validate the completeness and correctness of the requirements.

### 4.1 Formula Generation

There are multiple *output adapters* in ARSENAL, which convert the IR table (with the semantic metadata annotations) to different output forms. The current ARSENAL implementation includes adapters that convert the IR table to first-order logic (FOL) and linear temporal logic (LTL) formulas. In this paper, we discuss the SAL model adapter, which converts the IR table to a SAL model. The SAL model represents a transition system whose semantics is given by a Kripke structure, which is a kind of nondeterministic automaton widely used in model checking [CGP99].

ARSENAL uses Recursive Expression Translation (RET) rules to generate the SAL formulas from the predicate graph. For example, to generate an LTL formula from the predicate graph shown in Figure 9, we can employ the subset of RET rules in Table 2.

We use  $e(X)$  to denote the expression associated with mention  $X$ , which we will repeatedly rewrite during the translation process. If mention  $X$  is associated with a unique term, i.e.  $unique(X)$ , its expression is simply the English word in the mention, e.g.,  $e(Invalid-8) = Invalid$ . Given a predicate graph, we recursively apply the translation rules starting from the root (set-14 in this case). The superscripts  $u$ ,  $m$  and  $l$  indicate the types of the translation rule for unique terms, simple arithmetic expressions and logical expressions respectively. When multiple rules are applicable to the same mention, they are applied in the order of  $tr^l$  followed



**Fig. 9.** Predicate graph after the application of type rules.

**Table 2.** Partial list of Recursive Expression Translation Rules

Predicate	Expression Translation
$unique(X)$	$tr^u(e(X)) : e(X)$
$of(X, Y)$	$tr^l(e(X)) : tr^m(e(Y)).tr^m(e(X))$
$set(X) \wedge arg1(X, Y) \wedge arg2(X, Z)$	$tr^m(e(X)) : tr^u(e(Y)) = tr^u(e(Z))$
$equal(X) \wedge arg1(X, Y) \wedge arg2(X, Z)$	$tr^m(e(X)) : tr^u(e(Y)) = tr^u(e(Z))$
$impliedBy(X, Y)$	$tr^l(e(X)) : tr^m(e(Y)) \rightarrow tr^m(e(X))$

by  $tr^m$  and then  $tr^u$ . The resulting LTL formula after applying the translation rules is shown below. The recursive expression translation is equivalent to running a DFS traversal on the predicate graph with emissions.

$G ((Lower\_Desired\_Temperature.Status\_attribute = Invalid) \Rightarrow Regulator\_Interface\_Failure = TRUE)$

Every requirement is considered as a *global* requirement except when certain indicative words such as “initialize” are present in the sentence. Hence, we have the **G** operator in front quantifying it for all computations.

Once the output formula is created from the IR table, ARSENAL also reports any disconnected nodes detected by DFS to the end-user, since these nodes will not be processed by the translation algorithm (and hence not be part of the output formula). This approach is quite useful for debugging missing mentions (if any) in the output formula.

## 4.2 Consistency Analysis with Büchi Automata

We do initial formal analysis of the generated formulas using [Büchi automata](#). A Büchi automaton extends a finite automaton to infinite inputs. It is an  $\omega$ -automaton  $A = (Q, \Sigma, \delta, q_0, F)$  that consists of the following components:

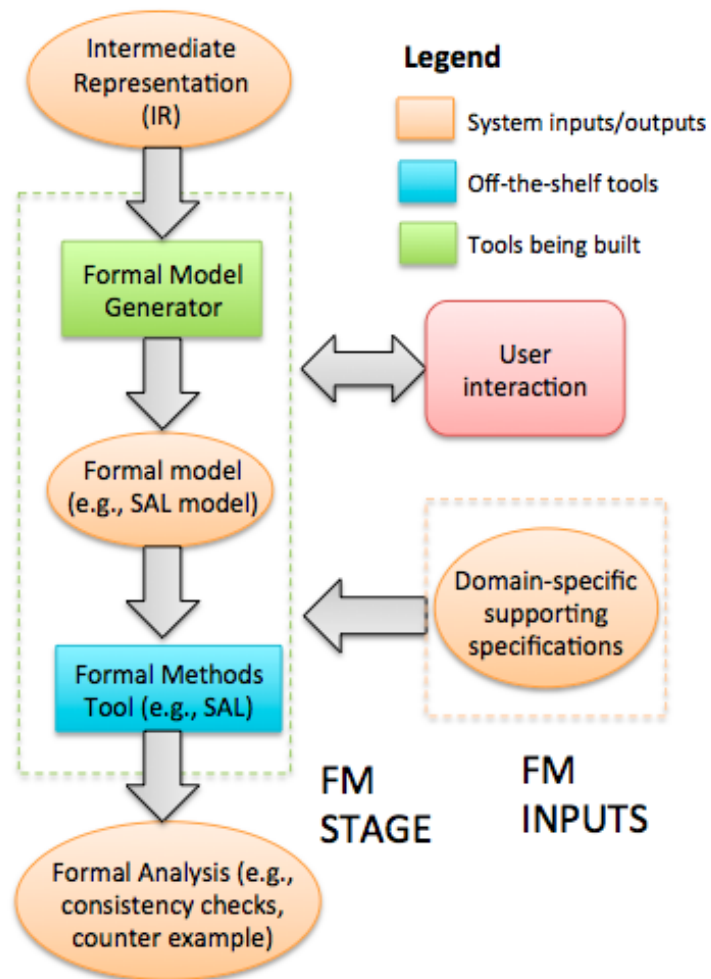


Fig. 10. FM Stage of ARSENAL pipeline.

- $Q$  is a finite set, whose elements are called the states of  $A$ .
- $\Sigma$  is a finite set called the alphabet of  $A$ .
- $\delta : Q \times \Sigma \rightarrow Q$  is a function, called the transition function of  $A$ .
- $q_0$  is an element of  $Q$ , called the initial state.
- $F \subseteq Q$  is the acceptance condition.  $A$  accepts exactly those runs in which at least one of the infinitely often occurring states is in  $F$ .

Büchi Automata is used in [model checking](#), as an automata-theoretic version of a formula in LTL. If the Büchi Automaton generated from a set of formulas is degenerate (i.e., empty), the formulas are inconsistent. Given a set of LTL formulas generated by the Formula Generator, we first check if the Büchi Automaton created from the corresponding propositionalized formulas is degenerate — if so, we report the inconsistency in the requirements to the user. If not, we proceed to creating the SAL model, as shown in Figure 12.

### 4.3 The SAL Modeling Language

In this paper, we focus on transition systems modeled using the SAL language [\[BGL<sup>+</sup>00\]](#). At the core, SAL is a language for specifying transition systems in a compositional way.

The semantics of a SAL transition system is given by a Kripke structure. A Kripke structure over a set of atomic propositions  $AP$  is a tuple  $\langle Q, Q_0, R, L \rangle$ , where  $Q$  is the set of states,  $Q_0 \subseteq Q$  is the set of initial states,  $R \subseteq Q \times Q$  is the transition relation, and  $L : Q \rightarrow 2^{AP}$  is a labeling function that assigns each state to a set of  $AP$  that is true in that state. Kripke structure is a kind of nondeterministic automaton that is widely used in model checking [\[CGP99\]](#).

```
sal_example : CONTEXT =
BEGIN
  Type1 : TYPE = [# high : BOOLEAN, low : BOOLEAN #];
  main : MODULE =
  BEGIN
    INPUT add: BOOLEAN
    LOCAL count: INTEGER
    OUTPUT out Type1
    DEFINITION
      out.high = count > 100;
      out.low = count < 50;
    INITIALIZATION
      count = 0;
    TRANSITION
      [ add = TRUE --> count' = count + 1
        []
        ELSE --> count' = count ]
  END;
  THEOREM main |- G (NOT(out.high = TRUE));
END
```

**Fig. 11.** Example SAL Model.

An example of a SAL program is given in Figure 11. This model represents a transition system whose state space is defined by the local variable “count”. Whenever input “add” is **true**, “count” is incremented by 1 (the apostrophe on “count” indicates it is the next state of “count”), otherwise, its value remains unchanged. This model also has a single output “out” having a record type. If “count” is greater than 100, the “high” field is set to **true**, and if “count” is less than 50, the “low” field is set to **true**. In this example, “main” is the name of a module. For simplicity, we consider only self-contained modules in this paper. “Type1 ...” is a type expression that creates “Type1” as a record type. In Section 3, we will describe how the type information is automatically inferred by ARSENAL.

The other relevant SAL language constructs for defining a transition system are “DEFINITION”, “INITIALIZATION” and “TRANSITION”. The “DEFINITION” section describes constraints over the *wires* of a module. The “INITIALIZATION” section gives the initial conditions of the *controlled* variables. Finally, the “TRANSITION” section constrains the possible next states for the *state* variables. In this example, the update of “count” is given by the guarded command with the guard “add = TRUE” and the command “count’ = count + 1”, where the apostrophe indicating it is referring to the next state of “count”. We refer the readers to [BGL<sup>+</sup>00] for a detailed description of the language. In general, SAL models are expressive enough to capture the transition semantics of a wide variety of source languages. One key contribution of ARSENAL is an automatic way of translating from the NL description of a transition system and its requirements directly to a SAL model and theorems.

#### 4.4 Verification with SAL

SAL can be used to prove theorems, encoding properties about the requirements, using bounded model-checking. If SAL finds a counter-example, we know the property does not hold. If SAL does not find a counter-example at a known depth of model-checking, we try to see if the LTL formulas are realizable.

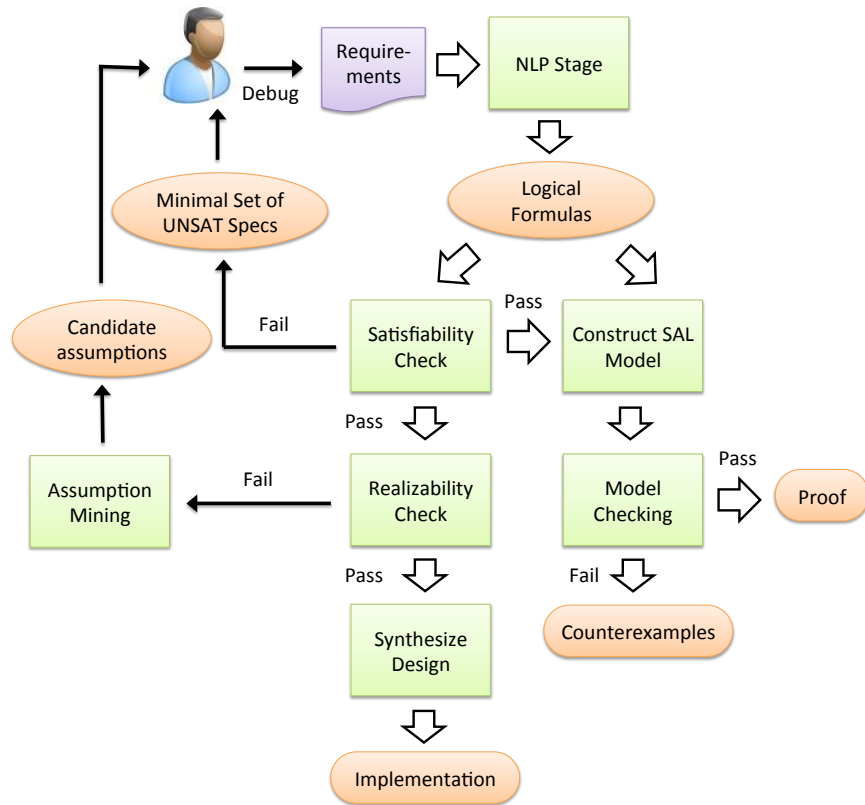
We continue to use REQ1 to illustrate how ARSENAL produces a SAL model from the generated formulas in the previous step. At its core, SAL is a language for specifying transition systems in a compositional way. A transition system is composed (synchronously or asynchronously) of modules, where each module consists of a *state type*, an *invariant definition* on this state type, an *initialization condition* on the state type, and a *binary transition relation* on the state type. The state type is defined by four pairwise disjoint types of variables: *input*, *output*, *local* and *global* — input and global variables are observed variables of a module, while output, global and local variables are controlled variables of a module. Note that the SAL model-checkers use Linear Temporal Logic (LTL), a modal temporal logic with modalities referring to time, as their underlying assertion language. This is an appropriate language for formally expressing requirements, since many requirements have temporal operators (e.g., eventually, always).

In order to unambiguously define a transition system, we need to additionally distinguish *controlled* variables that are *state* variables from *wires* (variables that do not directly define the state space). We need to know the type of any variable. Since we consider only self-contained modules in SAL, a variable can then belong to one of the following five categories: *input*, *state only*, *state and output*, *output only*, and *wire*. By differentiating *state* variables from *wires*, we can unambiguously map them to the corresponding section in SAL, namely DEFINITION or TRANSITION. We use the former to describe constraints over wires, and the latter to describe evolutions of state variables. We generate different types: INTEGER, enumerated types, and record types.

A snippet of the generated SAL model is shown below.

```
model_1 : CONTEXT =
BEGIN
  Type1 : TYPE = {Invalid};
  Type3 : TYPE = [# Temp_attribute : INTEGER, Status_attribute : type60 #];

  main : MODULE =
  BEGIN
    LOCAL Regulator_Interface_Failure : BOOLEAN
    LOCAL Lower_Desired_Temperature : Type3
    DEFINITION
      Regulator_Interface_Failure IN {Z : BOOLEAN |
        Lower_Desired_Temperature.Status_attribute = Invalid
        => Z = TRUE};
    END;
  END
```



**Fig. 12.** FM Flow in ARSENAL pipeline.



Now, if the `Regulator_Interface_Failure` variable in REQ1 was a *state* variable, then the SAL model generator would have produced the following TRANSITION instead of the DEFINITION.

TRANSITION

```
Lower_Desired_Temperature.Status_attribute = Invalid
--> Regulator_Interface_Failure' = TRUE
```

The SAL model would hence be different, even though generated from the same sentence. Currently, ARSENAL requires the user to provide this additional semantic information only after the NLP stage, thus keeping it separate from the model-independent part of the pipeline.

**Table 3.** Rules for Gathering Type Evidence

Expression	Inference
$X \bowtie Y, \bowtie \in \{<, >, \leq, \geq\}$	$X$ and $Y$ are numbers
$X$ is a number	$X$ is a number
$X$ is a named value $C$	$X$ has enum type containing $C$
$X = Y$	$X$ and $Y$ have same type

During the model generation stage, ARSENAL gathers type evidences for each variable across all sentences, and performs type inference by merging them into equivalence classes. Further, in case of a type conflict, a warning is produced to indicate inconsistency in the NL sentences, thus helping the user to refine their requirements documentation at an early stage. Table 3 summarizes the rules ARSENAL currently implements for gathering type evidence. Figure 13 outlines the algorithm for merging types into equivalence classes.

```
for each type equivalence class C:
  Type T = merge(C)
  For each term X in C:
    Set the type of X to T

merge(R1 : RecordType, R2 : RecordType) =
  for each field F of R2:
    TF1 = the type of F in R1
    if R1 does not have field F:
      Add field F to R1
    else:
      TF2 = the type of field F in R2
      merge(TF1, TF2)
  return R1

merge(X, X): return X when X is Bool or Integer
```

**Fig. 13.** Algorithm for Type Merging in ARSENAL.

We automatically determine where each formula goes using the following heuristics, and populate the SAL model accordingly:

- If the formula contains temporal operators (explicitly), it is a theorem.
- If the formula has an assignment to a state variable, it is a transition.
- If the top-level IR entry (root) that generated the formula is “initialize”, it is an initialization.
- Otherwise, it is a definition, i.e. assignment to a local wire.

## 4.5 Synthesis with RATS

Given an LTL specification, it may also be possible to directly *synthesize* an implementation that satisfies the specification. *Realizability*, the decision problem of determining whether such an implementation exists, can be used to further inspect the requirements document for inconsistencies. If the specification is realizable, then a Moore machine can be extracted as an implementation that satisfies the specification. Thus, the benefit of LTL synthesis is a correct-by-construction process that can automatically generate an implementation from its specification. In general, LTL synthesis has high computational complexity [Kup12]. However, it has been shown that a subclass of LTL, known as Generalized Reactivity (1) [GR(1)], is more amenable to synthesis [PP06] and is also expressive enough for specifying complex industrial designs [BGJ<sup>+</sup>07]. Given input/output partitions of the Boolean variables into  $I \cup O$ , GR(1) specification takes the form  $\phi_e \rightarrow \phi_s$ , where each  $\phi_i$  is a conjunction of the following:

- Initial states:  $\alpha_i$ , which is a Boolean formula that characterizes the initial states of the environment/system.
- Transitions:  $\beta_i$ , which is a formula of the form  $\bigwedge_j \mathbf{G} f_j$  where each  $f_j$  is a Boolean combination of variables from  $I \cup O$  and expressions of the form  $\mathbf{X} v$  where  $v \in I$  if  $i = e$ , and  $v \in I \cup O$  otherwise.
- Fairness:  $\gamma_i$ , which is a formula of the form  $\bigwedge_j \mathbf{G} \mathbf{F} f_j$ , where each  $f_j$  is a Boolean formula.

Given a LTL specification  $\psi$  that is satisfiable but not realizable, the assumption mining problem is to find  $\psi_a$  such that  $\psi_a \rightarrow \psi$  is realizable. Our algorithm for computing  $\psi_a$  follows the counterstrategy-guided approach in [BCG<sup>+</sup>10], which has shown to be able to generate useful and intuitive environment assumptions for digital circuits and robotic controllers. The algorithm is based on [LDS11], and is summarized below.

**Counterstrategy-guided synthesis of environment assumptions.** Given the unrealizable specification  $\psi$ , the method first computes a counterstrategy. The counterstrategy summarizes the next moves of the environment in response to the current output of the system, which will force a violation of the specification. The method then uses a template-based mining approach to find a specification  $\phi$  that is satisfied by the counterstrategy.  $\neg\phi$  is added as a new conjunct to  $\psi_a$  and  $\psi_a \wedge \psi_e \rightarrow \psi_s$  is checked for realizability again. By asserting the negation of  $\phi$  as an assumption to the original specification, the method effectively eliminates the moves by the environment that adhere to the counterstrategy. The process iterates until the resulting specification becomes realizable. At any step of the iteration, the user is asked to verify the correctness of the mined assumption, as shown in Figure 12. We present them as NL sentences to the user, which we generate by mapping the boolean and temporal operators to English connectives. If the user rejects an assumption, another one (or set of) assumption is produced, until either the specification is realizable or all the recommendations are rejected by the user.

## 5 Evaluation

In this section, we present results on analyzing ARSENAL’s ability in handling complex NL sentences and different corpora. We measure the degree of automation of ARSENAL, its robustness to noise, and the accuracy of the NLP stage.

### 5.1 Degree of Automation Metric

In this section, we report automation results of ARSENAL on both the FAA-Isolette (FAA) and the TTether (TTE) corpora. Specifically, we evaluate the accuracy of ARSENAL’s NLP pipeline on translating each NL sentence into the corresponding logical formula automatically, without any manual correction. This metric measures the degree to which ARSENAL runs in an automated mode.

The results are summarized in Table 4. When evaluating accuracy, the correct outputs were given a score of 1.0, wrong outputs were given a score of 0.0, while partially correct results were given partial credit of

**Table 4.** ARSENAL NLP pipeline accuracy.

Corpus	Total	Correct	Partial	Wrong	Degree of Automation
TTE	36	24	8	4	78%
FAA	42	39	2	1	95%

**Table 5.** Results of perturbation test on ARSENAL.

Perturbation Type	TTEthernet domain (TTE)			FAA-Isolette domain (FAA)		
	Total sentences	Perturbed sentences	Accuracy	Total sentences	Perturbed sentences	Accuracy
First (And→Or)	36	16	81%	42	N/A	N/A
All (And→Or)	36	16	87%	42	13	92%
All (Is→Is not)	36	17	100%	42	13	92%
If A then B→B if A	36	N/A	N/A	42	40	65%

0.5. A translation was deemed partially correct if there was one error and incorrect if there was more than one error in the resulting formula.

Note that when ARSENAL fails to give the correct output automatically from the NLP stage while processing requirements, we correct the error manually so that the input to the FM stage is correct. The following sentence is one of the sentences in FAA for which ARSENAL partially captures the logical semantics.

**REQ:** *If the Regulator Mode equals NORMAL, the Temp attribute of the Display Temperature shall be set to Temp attribute of the Current Temperature rounded to the nearest integer.*

The logical formula output by ARSENAL is:

```
(Regulator_Mode = NORMAL => Display_Temperature.Temp_attribute =
Current_Temperature.Temp_attribute)
```

The reason ARSENAL only handles the first half of the sentence correctly is that the phrase “rounded to the nearest integer” implies there is a function that can take a real/floating-point number as input and produce its *nearest* integer as output. Currently, ARSENAL does not have support for arbitrary functions — in the future, we plan to incorporate more domain-specific knowledge and have built-in support for frequently occurring functions.

## 5.2 Degree of Perturbation Metric

We define an evaluation criteria for measuring the robustness of ARSENAL, i.e., if perturbations/modifications are made to a requirements sentence using certain rewrite rules, whether ARSENAL can still generate the right output formula.

For the given dataset (e.g., FAA or TTE), we do perturbations to the requirements in that dataset using a transformational grammar, having operators that transform the text. The transformations in this grammar are based on allowable terminals in SAL, e.g., we can replace “always” by “eventually”, “or” by “and”, “is” by “is not”, etc. By applying these transformation operators to the FAA dataset, we can generate a “perturbed” dataset. This is similar in principle to generating test cases by fuzz testing [GLM08]. Note that transforming “or” to “and” can be a significant perturbation for ARSENAL if a sentence has a combination of “and” and “or” terms, since the transformation can change the nesting structure of clauses in the output formula.

Table 5 shows the results of our experiments on the FAA and TTE datasets. Note that total number of requirements was 42 in FAA and 36 in TTE. Out of the 36 requirements in TTE, the “And → Or” rewrite rule

affected 16 requirements. We ran two types of “And  $\rightarrow$  Or” transformations — in the first case, we modified only the first occurrence of “And” in the requirements sentences, while in the second case we modified all occurrences of “And” in the sentences. When ARSENAL was run on these transformed requirements, thirteen of them gave output formulas that were correct w.r.t. the modified requirements sentence for the “First (And  $\rightarrow$  Or)” rewrite rule, while fourteen of them gave output formulas that were correct for the “All (And  $\rightarrow$  Or)” rewrite rule, giving an accuracy of  $13/16 \approx 81\%$  and  $14/16 \approx 87\%$  respectively. Similar numbers were calculated for other rules on FAA and TTE. For FAA, only 2 sentences had more than one AND in them — so we did not run the “First (And $\rightarrow$ Or)” transformation on FAA, since the results for that would have been quite close to the “All (And $\rightarrow$ Or)” rule.

We subsequently tried more complex rewrite rules, e.g., of the form “If A then B $\rightarrow$ B if A”. For the “If A then B $\rightarrow$ B if A” rule, ARSENAL’s lower accuracy of 65% on the FAA domain was mainly caused by incorrect parse output from STDP on the perturbed sentences. The most common cause of failure is “implication in the wrong place”, i.e., STDP doesn’t know the correct scope of the “if” and “then” clauses. Here is an example demonstrating this error:

**REQ:** *If the Regulator Mode equals INIT and the Regulator Status equals True, the Regulator Mode shall be set to NORMAL.*

The corresponding ARSENAL output is:

```
((Regulator_Mode = INIT AND Regulator_Status = TRUE) => Regulator_Mode = NORMAL)
```

**Modified REQ:** *The Regulator Mode shall be set to NORMAL, if the Regulator Mode equals INIT and the Regulator Status equals True.*

Corresponding ARSENAL output:

```
(Regulator_Mode = INIT => (Regulator_Mode = NORMAL AND Regulator_Status = TRUE))
```

The failure of ARSENAL to get the correct formula in this case is mainly because “A, if B and C” is interpreted by the Stanford parser to mean “(A if B) and C”, instead of “A if (B and C)”.

In general, writing requirements in active voice is better than requirements in passive voice, especially when A and B are complex clauses. Motivated by this perturbation analysis, we added a feature in ARSENAL — whenever a requirement is in passive voice, we detect that and issue a message to the user that active voice is preferred, since one of the goals of ARSENAL is to train the requirements engineer to write better requirements.

For TTE, none of the 36 sentences had the “If A then B” structure.

### 5.3 NLP Stage Accuracy

Our goal here was evaluating the accuracy of the NLP stage of ARSENAL. There are existing metrics to calculate the degree of overlap between two semantic feature structures, e.g., Smatch [CK13] uses ILP and hill-climbing. For ARSENAL, we took the more direct approach of estimating how many sub-formulas are inserted, deleted or modified by ARSENAL in the NLP stage, while generating the output formula for a requirements sentence. We first considered a corpus of requirements sentences that have been annotated with the expected output formula, which we call the *ground-truth corpus*. Then, we used the following algorithm to compute a novel metric:

1. Given each requirements sentence in this ground-truth corpus, generate all sub-formulas  $G$  of the ground-truth formula and all sub-formulas  $A$  of the formula generated by ARSENAL.
2. For each sub-formula in  $A$ , find the best-matching sub-formulas in  $G$  using a relaxed version of *Max-weighted matching in bipartite graphs*. The match score between formulas is calculated using Typed Levenshtein distance (a typed variant of the standard Levenshtein edit distance that we designed), suitably

modified for formulas to consider distances between logical symbols, variables and string tokens differentially.

3. Calculate precision, recall and F-measure using the similarity scores between matched pairs of sub-formulas.

**Typed Levenshtein Distance** This is a modified version of the Levenshtein string edit distance, where a higher-level token-based edit distance calls an underlying character-based edit distance. The computation of that score is outlined below:

```
d[i][j] =
  if seq1[i] == seq2[j]
    then d[i+1][j+1]
  else min(
    IC + d[i+1][j],
    DC + d[i][j+1],
    c(seq1[i], seq2[j]) + d[i+1][j+1])
```

Here,  $d$  is the distance, IC is insertion cost, DC is the deletion cost,  $seq1$  and  $seq2$  are the two sequences, and  $c(A, B)$  is the cost of replacing  $A$  by  $B$ . We first turn the formula into a sequence of tokens, with 3 types of tokens:

1. LogicalSymbol Token (e.g., “and”, “or”, “exists”):  $c(A, B) = 1$  if  $A \neq B$ .
2. String Token (e.g., argument name):  $c(A, B) = \text{LevenshteinStringEditDistance}(A, B)$ .
3. Variable Token:  $c(A, B) = 0$ , since variable names can change, and we do not consider any cost for that.

**Max-weighted matching in bipartite graph** We next form a bipartite graph, where the sub-formulas of  $G$  and  $A$  are the nodes, and the edge connecting each node in  $G$  to each node in  $A$  is the Typed Levenshtein distance between those 2 formulas.

Max-Weight Bipartite Matching is NP-Hard — so, we relax the 1:1 matching constraint in max-weight bipartite matching to get an efficient algorithm. Once we get the matching, we use that to compute the precision, recall and F-measure of the matching.

Figure 14 shows the bipartite graph for an example pair of formulas after matching, where the nodes correspond to the sub-formulas and the edges correspond to the Typed Levenshtein distance between the best matched sub-formulas. The figure also shows the F-measure calculation for this graph.

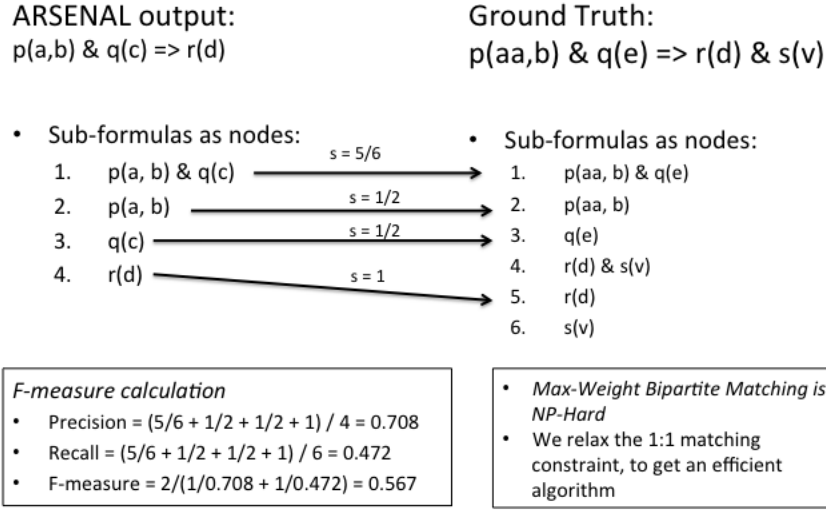
**Evaluation on Test set** We evaluate the F-measure of the NLP stage of ARSENAL on sections in the TTEthernet Requirements [SD10] and standards documents. We also ran the evaluation on requirements from the CPGA, LTLMap and Eurail requirements documents (details in Section 1). For each requirements document, we trained ARSENAL (e.g., tuned the preprocessing rules) on some of the sentences, and evaluated the F-measure only on the hold-out test set on which ARSENAL was not trained in any way. Considering the different corpora, there were overall 300 sentences — of these, we used 112 sentences as the training set for tuning ARSENAL, and evaluated the performance of ARSENAL on the remaining 188 sentences that were held out. Table 6 shows the results — on the test set of 188 requirements, the NLP stage had an overall F-measure of 0.63.

Here are some of the errors made by ARSENAL:

TTEthernet requirements:

**REQ:** *A supported write access to the AMBA 3 AHB-Lite V1.0 slave interface shall be processed and terminated with response OKAY.*

There is no TD between “write” and “access”, so everything that follows further down that dependency tree is missing from the generated formula. This can be fixed by adding a preprocessing entry that treats “write access” as an n-gram in this domain.



**Fig. 14.** Bipartite graph and F-measure calculation corresponding to example formula and ARSENAL output.

**Table 6.** NLP stage F-Measure on different corpora.

Corpus	Test set size	Test set F-measure
TTEthernet	72	0.65
Eurail	11	0.58
GPCA	70	0.62
LTLMop	35	0.64
Total	188	0.63

**REQ:** If the *ct\_mode* entry (in the CT Table) of a CT ID provided at a critical frame output interface is set to anything but OUT, layer 3 shall set the *Fo\_State\_Im* flag of the *Fo\_State* field and drop the current transfer on that interface.

The phrase “anything but”, another way of stating a negation, it not currently handled by ARSENAL — so the output formula misses a sub-formula.

Eurail requirements:

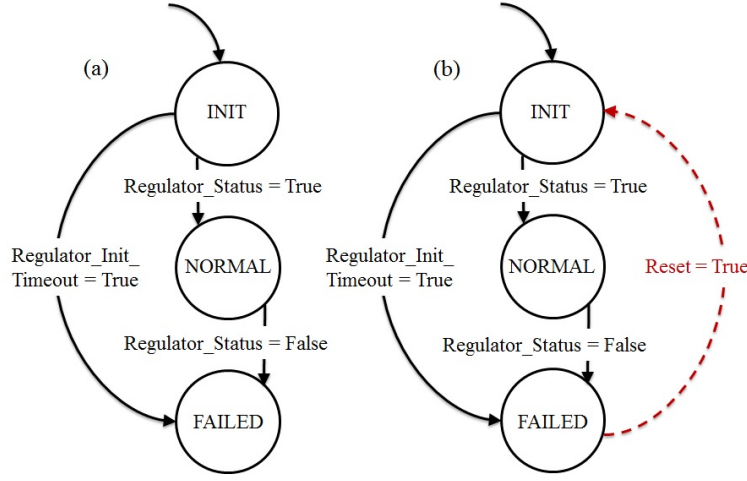
**REQ:** A Radio Infill Unit (see section 3.9.3) shall never initiate a communication session.

The negation is missing from the generated ARSENAL formula, because our type rules interpret “never” as a temporal attribute rather than a normal negation. We can extend type rules to understand “never” as negation in the correct context.

#### 5.4 Case Study: FAA-Isolette

We also did detailed case studies of the FAA-Isolette and TTEthernet requirements. Figure 15 (a) shows one of the finite state machines corresponding to the regulator function of the isolette. An example requirements sentence is shown below.

**REQ:** *If the Regulator Mode equals INIT and the Regulator Status equals True, the Regulator Mode shall be set to NORMAL.*



**Fig. 15.** Original FSM (a) and Modified FSM (b) for Regulator.

This experiment seeks to evaluate if ARSENAL can faithfully generate the transition system corresponding to the description (including the FSM) in the design document. Verification was used to validate the generated FAA model, corresponding to the following SAL theorem generated by ARSENAL:

```
THEOREM main |- G((Regulator_Mode=FAILED => NOT(F(Regulator_Mode=NORMAL))));
```

This theorem states that if the FSM is in the FAILED state, then it cannot go back to the NORMAL state (F in the theorem means *eventually*). Applying model checking, we verified that the generated SAL model satisfied the theorem.

To demonstrate the applicability of ARSENAL in identifying inconsistencies in NL requirements, we added a sentence corresponding to the transition from the FAILED state to the INIT state, as shown in Figure 15 (b). For the modified model, ARSENAL quickly produced a counterexample that showed a path from the FAILED state to the NORMAL state, thus violating the aforementioned theorem. This demonstrates that the SAL model generated automatically by ARSENAL behaves as expected.

Application of LTL [GR(1)] synthesis to these formulas produced an unrealizable result; no Moore machine existed to satisfy the formulas. ARSENAL produces the following candidate assumption to make the specification realizable.

```
G !(Regulator_Status=1 & Regulator_Init_Timeout=1);
```

To better understand why this assumption is necessary for ARSENAL to generate a SAL model from the set of sentences, observe that in Figure 15, the INIT state has two outgoing transitions, one to the NORMAL state and the other to the FAILED state. When both `Regulator_Status` and `Regulator_Init_Timeout` are `true`, the state machine can nondeterministically choose to go to either of the states. Such behavior is not desirable in an actual implementation, which is supposed to be deterministic. Hence, the original specification is not realizable. Once the NL sentences describing the transitions are written differently, such that `Regulator_Status=1` and `Regulator_Init_Timeout=1` are mutually exclusive, the specification becomes realizable and an implementation can be generated automatically, e.g., in Verilog.



## 6 Related Work

We first give an overview of the research related to ARSENAL.

### 6.1 Summary of Related Work

There is a rich and diverse body of research related to requirements engineering. The main advantages of ARSENAL over prior work are a less restrictive NL front-end, a more powerful FM analysis framework, and a stronger interaction between the NL and FM stages.

Kress-Gazit et al. [KGFP08], Smith et al. [SACO02] and Shimizu et al. [Shi02] propose grammars for representing requirements in controlled natural language. The natural language interfaces suggested in these papers are restrictive — ARSENAL can extract information automatically from a wider range of natural language styles.

Zowghi et al. [ZGM01], Gervasi et al. [GZ05], Scott et al. [SCK04], Xiao et al. [XPTX12], Ding et al. [DJP11], and Post et al. [PH12] process constrained natural language text using NLP tools (e.g., CFG, Cico) and perform different types of checks (e.g., consistency, access control) on the requirements. Compared to these methods, ARSENAL uses more state-of-the-art NLP techniques that can be made domain-specific using resources like the domain-specific ontology, customized regex-based template matching in pre-processing, etc. ARSENAL also uses more advanced model checking tools (e.g., SAL), which can represent theorems and invariants in a much more expressive logic (e.g., LTL).

Behavior-driven development (BDD) is a way to give natural language specifications for the software testing phase. Drechsler et al. [DDG<sup>+</sup>12], Soeken et al. [SWD12], and Harris [Har12] show different ways of translating high-level natural language requirements to tests in the BDD framework, which can then be used by BDD tools like Cucumber [cuc] or RSpec [rsp]. ARSENAL is more general than these approaches — instead of considering requirements specifications at the test phase, it considers NL requirement specifications that can be specified and verified in the design phase.

Ormandjieva et al. [OKH07], QUARS [FFGL01], and Goguen [Gog96] focus on assessing the quality of requirements documents — they do not create formal models from the requirements for downstream formal analysis (e.g., consistency checks) like ARSENAL. Malin et al. [Mal09], Boyd [Boy99], and Nikora et al. [NB09] do linguistic modeling and information extraction from requirements documents, but do not handle consistency checks or downstream formal methods analysis (e.g., using SAL) like ARSENAL. Attempto Controlled English (ACE) [SF96], RECORD [Bor96] and T-RED [BS96] are user-oriented tools for requirements collection, reuse, and documentation. These are interactive tools requiring inputs from domain experts, and are not as automated as ARSENAL.

### 6.2 Details of Related Work

We next discuss details of related research in two major areas relevant to our work — requirements engineering and natural language processing — and highlight the comparative advantage of ARSENAL.

**Requirements Engineering** We begin with a comparison of ARSENAL to two approaches in requirements document analysis.

**PROPEL:** In PROPEL [SACO02], the authors note that properties used in formal verification map to one of several property pattern templates. These pattern templates are shown to the user in an interactive framework as choosing among these options should help the specifier consider the relevant alternatives and subtleties associated with the intended behavior.

The authors represent these pattern templates using two different notations: an extended finite-state automaton (FSA) representation and a disciplined natural language (DNL) representation. The DNL representation provides a short list of alternative phrases that highlight the options, as well as synonyms for each option to support customization. This representation should appeal to those specifiers who prefer a natural



language description, and do not want to handle the full rigor of formal property specifications. The extended FSA representation provides a graphical view that can be used to derive a specific FSA representation.

This work differs with our approach in a few important ways:

1. The authors do not extract property templates automatically from given natural language specifications. Instead, they provide a tool that facilitates the user in writing specifications, and gives tips to modify the specification to make it less ambiguous. ARSENAL can extract information automatically from given natural language specifications, and does not have to rely solely on the user to modify specifications like PROPEL does.
2. ARSENAL handles more general input sources, e.g., specs from tables, flow charts.
3. ARSENAL also has a mechanism for checking consistency across multiple specifications, which is a generalization of PROPEL.

**CARL:** Zowghi et al. [ZGM01] address the issue of finding inconsistencies in requirement documents. They represent requirements as constrained natural language text and convert these specifications into formulas in propositional logic using an NLP parser, Cico. Then they use a reasoning system, CARET, to analyze the inconsistency in specifications, where requirements are modeled as default theories. The authors demonstrate the effectiveness of their approach using data from the London Ambulance System (LAS), which uses a computer-aided dispatch system. They evaluate this case study in detail and find inconsistencies in particular requirements specifications in LAS.

Some of the key advantages of ARSENAL over CARL are:

1. CARL uses a simple NLP tool called Cico, which parses the text to get entities and relations that are used to get the formulas directly. ARSENAL uses more state-of-the-art NLP techniques, which can be made domain-specific using resources like the glossary, customized regex-based template matching, and so on.
2. Consistency engine (CARET) uses classical logic and non-monotonic reasoning. The underlying logic used by the formal methods approaches in CARL is default theory, which uses deterministic propositional formulas. In comparison, ARSENAL uses more advanced model checking tools (e.g., SAL).

## Other Research

Some other notable related research activities in requirements engineering are discussed next, each of which has technology related to some parts of the ARSENAL pipeline.

1. Gervasi et al. [GZ05] explore the integration of natural language parsing techniques with default reasoning to overcome the difficulty of training system designers directly using logic. They also propose a method for automatically discovering inconsistencies in the requirements from multiple sources, using both theorem-proving and model-checking techniques. The effectiveness of their approach and tool is illustrated on an example domain involving conflicting requirements. Scott et al. [SCK04] have proposed a context-free grammar to facilitate the parsing of requirement text, e.g., finding temporal clauses, condition clauses, relative clauses. The grammar is used to drive an interactive tool that takes the specification input by a user and standardizes it to the predefined format specified by the grammar, prompting the user to accept the re-formatting. The structured requirements can then be used by case-based reasoning systems for comparison and consistency checks.

While the above systems had goals similar to those of ARSENAL, the latter uses more advanced and state-of-the-art NLP, machine learning and formal methods tools and can therefore handle more complex (potentially ambiguous) specifications of different types (from multiple sources).

2. Ormandjieva et al. [OKH07] propose a quality model for evaluating requirements text, and a text classification system to automate the quality assessment process. The text classifier was trained on features that were extracted automatically from the text (output of a Part-of-Speech tagger and a parser), with human judgment as ground truth. The classifier was able to actually flag ambiguous and unambiguous texts at the surface level of understanding. They ran a large study using human raters. Inter-rater

agreement levels showed that the quality assessment task is difficult for humans, but there is reasonable agreement on the chosen quality indicators, both at the level of surface understanding and at the level of conceptual understanding. The study also demonstrated that automatic detection of ambiguities in requirements documentation has good performance, comparable to human judgment. Other relevant work in this area is QUARS by Firing et al. [FFGL01], a tool for automatic quality evaluation for natural language requirement. Goguen [Gog96] also discusses the trade-offs between formal representation and informal specifications in requirements documents.

The above papers focus only on assessing the quality of requirements documents or predicting the judgment of a human user about the quality of the requirements. In contrast, ARSENAL is much broader in scope.

3. The Automated Tool and Method for System Safety Analysis project [Mal09] used a Semantic Text Analysis Tool (STAT) to extract key information from failure modes and effects analyses (FMEAs) and hazard reports. The primary sources of extractions are FMEA documents, which contain system descriptions, problem descriptions, and statements about connections and dependencies. Model generation software in the Hazard Identification Tool (HIT) integrates this information into visualizations of system architecture models. The intent is to make it easier to review hazard paths and find redundant and missing links within and between types of analysis. Boyd [Boy99] describes a controlled natural language that can be used to specify software development models.

The above papers do linguistic modeling and information extraction from requirements documents, but do not handle consistency checks or downstream formal methods analysis like ARSENAL.

4. Crow et al. [CV96] use formal methods tools (PVS, Murphy) for case studies on the flight-software subsystem in the NASA shuttle program, using formal specification techniques or using state exploration approaches. The key technical results of the paper are a clear demonstration of the utility of formal methods, not as a replacement but as a complement to the conventional Shuttle requirements analysis process. The application of formal methods to the particular projects considered in this paper – JS, 3E/O, and GPS – each uncovered anomalies ranging from minor to substantive, most of which were undetected by existing requirements analysis processes. The main insight from the paper is that formal methods techniques are most effective when they are judiciously tailored to the application.

This work does not have an end-to-end system involving information extraction, consistency checking and formal verification like ARSENAL – it focuses only on the formal methods part.

5. Another domain-specific language is Attempto Controlled English (ACE) [SF96], a subset of English with a restricted grammar and a domain-specific vocabulary, which allows domain specialists to interactively formulate requirements specifications in domain concepts. RECORD [Bor96] and T-RED [BS96] are user-oriented tools for requirements collection, reuse, and documentation.

These are interactive tools requiring inputs from domain experts and are not fully automated like ARSENAL.

6. Shimizu et al. [Shi02] discuss how formal specification can be written from natural language requirements, for commonly used interface protocols, and how test inputs and the checking properties can be generated automatically from the formal specification.

However, unlike ARSENAL, the above work does not involve automatic extraction of formal specifications from natural language.

**Natural Language Processing (NLP)** Here we discuss related work corresponding to the NLP part of the ARSENAL pipeline, which does information extraction from semi-structured text. This problem has been studied in different domains, e.g., clinical text [dBCK<sup>+</sup>11], legal documents [SNM10], web tables [CP10]. For example, the authors in [SNM10] have an approach where only relevant text segments (e.g., corresponding to litigation claims) are extracted from a full legal document, and then relevant entities (e.g., patents, laws) are extracted from those text segments. In ARSENAL we extract *both* entities and relations using NLP tools, and additionally connect their output to Formal Methods tools to facilitate more detailed downstream analysis.

Within the NLP portion of the ARSENAL pipeline, another core component is the extraction of text spans to fill the slots in the template structure. The following related work also use the general idea of breaking documents into text spans for further processing:

- Teufel et al. [TM02] present an approach to summarizing scientific articles that is based on the idea of restoring the discourse context of extracted material by adding the rhetorical status to each sentence in a document. The innovation of their approach is that it defines principles for content selection specifically for scientific articles, and that it combines sentence extraction with robust discourse analysis. The output of their system is a list of extracted sentences along with their rhetorical status (one of seven rhetorical categories). The output of the proposed extraction and classification system can be viewed as a single-document summary. In addition, it provides starting material for the generation of task-oriented and user-tailored summaries designed to give users an overview of a scientific field. The authors present several experiments measuring the agreement of human judges with the system output, on the rhetorical annotations.
- Heilman et al. [HS10] present an algorithm for extracting simplified declarative sentences from syntactically complex sentences. They motivate their extraction approach by issues that are relevant for automatic question generation. The authors extract simplified sentences from complex linguistic structures in documents, e.g., appositives, subordinate clauses. They use the extracted simple directives to automatically generate questions based on the input text. Their system successfully simplifies sentences and transforms them into questions. They evaluate the approach using experiments and show that it is more suitable for extraction of factual question generation than a standard text compression baseline, which takes as input a possibly long and complex sentence and produces as output a single shortened version to convey the main piece of information in the input.
- Barker et al. [BCC<sup>+</sup>04] present a question-answering system that was developed as part of the HALO project, and discuss the results of its evaluation. The system was developed using a combination of several knowledge representation and reasoning technologies, in particular semantically well-defined frame systems, automatic classification methods, reusable ontologies, and a methodology for knowledge base construction. The system was able to encode the knowledge from 70 pages of a college-level chemistry textbook into a declarative knowledge base, and successfully answer questions comparable to questions on an Advanced Placement exam with high accuracy. In addition, the authors extended existing explanation generation methods, allowing the system to produce high-quality English explanations of its reasoning. The resulting system, in addition to having high accuracy, gave explanations that are comparable in quality to human judgments on tests.

Related work in semantic parsing includes PropBank [KP03], which annotates text with specific semantic propositions — ARSENAL has a rich intermediate representation for domain-targeting to multiple output languages. SEMAFOR [DSCS10] learns semantic frames per sentence, whereas ARSENAL generates a complete model directly from multiple sentences with appropriate semantics. Liang et al. [LJK11] introduce a new semantic representation, dependency-based compositional semantics (DCS), for learning to map questions to answers via latent logical. The predicate graph representation used in ARSENAL is more general than DCS, as it has both mention-level and relation-level semantic annotations. Chen et al. [CM11] focus on parsing natural language instructions to get a navigational plan for routing, and learn a semantic parser for the task; Walter et al. [WHH<sup>+</sup>13] have a framework for learning generalized grounding graphs for learning semantic maps from natural language descriptions. ARSENAL does not currently learn the semantic processor, but it can handle a more general class of output models, namely, SAL models encoding LTL logic.

In ARSENAL, since we have a controlled vocabulary for the domain and the underlying text is semi-structured, we propose to use a regular expression (regex)-based text span extraction technique — we take this simple but flexible approach, so that the regex matching rules can be updated effectively, and learned from data if necessary.

**Compliance checking and monitoring** There are other compliance checking and monitoring tools in the privacy and security domains. The HIPAA Compliance Checker [BDMN06] is a formal translation of

HIPAA into Prolog – the current implementation checks compliance with HIPAA rules, doing goal-driven evaluation of privacy policies and developing automated support for privacy policy compliance and audit. Basin et al. [BKM10] monitor complex security properties using a runtime approach using metric first-order temporal logic, and experimentally evaluate the performance of the resulting monitors. These approaches do not generate the logic formulas directly from the natural language specifications, which is the task that ARSENAL focuses on.

## 7 Conclusion and Future Work

In this paper we have presented ARSENAL, a methodology for generating complete formal models from natural language requirements text. The empirical results across requirements from multiple domains provide concrete empirical evidence that it is possible to bridge the gap between natural language requirements and formal specifications, achieving a promising level of performance.

The key accomplishments of ARSENAL are outlined in Figure 16.

- |  |
|--|
| <ol style="list-style-type: none"> <li>1. <b>Creating an NLP workflow for generating the IR:</b> <ol style="list-style-type: none"> <li>a) ARSENAL does semantic parsing using the combination of a type dependency parser, metatags and type rules,</li> <li>b) Resolves co-references and ambiguities in complex requirements sentences,</li> <li>c) Handles both domain-independent (e.g. for arithmetic expressions) and domain-specific pre-processing.</li> </ol> </li> <li>2. <b>Creating a FM workflow to generate a complete SAL model from the IR:</b> <ol style="list-style-type: none"> <li>a) ARSENAL has multiple output generators, to generate the appropriate output (e.g., FOL formula, SAL model) for a domain.</li> <li>b) For SAL model generation from IR, ARSENAL               <ol style="list-style-type: none"> <li>(i) Has principles to determine which formula should go to which part of the SAL model automatically.</li> <li>(ii) Automatically determines the SAL types, when the user only provides the input type categories.</li> <li>(iii) Guides the user to come up with the right formulation of the FM theorem, in natural language.</li> <li>(iv) Provides a debugging environment to the FM expert, helping to discover missing assumptions in the text.</li> </ol> </li> <li>c) ARSENAL generates counter-examples, constructs proofs of properties, and uses realizability to check inconsistency of requirements.</li> </ol> </li> <li>3. <b>Connecting the NLP and FM stages to create an end-to-end pipeline for both FAA-Isolette and TTEthernet domains:</b> <ol style="list-style-type: none"> <li>a) ARSENAL was developed on the FAA domain and later ported to the more complex TTEthernet domain,</li> <li>b) Has a modular design that helped isolate the parts that needed to be changed (e.g., pre-processor) without modifying the core parts,</li> <li>c) Has many algorithms (e.g., applying type rules) that are quite robust to porting to a new domain.</li> </ol> </li> <li>4. <b>Designing novel evaluation metrics to assess the performance of ARSENAL</b> (detailed numbers in Section 5.1):       <ol style="list-style-type: none"> <li>a) ARSENAL is automated to a large degree (as measured by the degree of automation metric),</li> <li>b) Is robust to requirements perturbation (as measured by the degree of perturbation metric),</li> <li>c) Has good accuracy in the NLP stage in generating the output formula on test datasets (as measured by a novel accuracy metric).</li> </ol> </li> <li>5. <b>Saving significant development cycles of the end-user:</b> <ol style="list-style-type: none"> <li>a) ARSENAL creates a first-cut formal model automatically from voluminous requirements, manually creating which requires a significant effort,</li> <li>b) Needs the user input to be provided only once per application domain,</li> <li>c) Allows user training efforts in formal modeling to be minimized.</li> </ol> </li> </ol> |
|--|

**Fig. 16.** Key accomplishments in ARSENAL.

In the future, we would place primary emphasis on making the ARSENAL framework more robust. We want to test ARSENAL on multiple other domains and datasets, and design more evaluation metrics like the ones discussed in this paper (e.g., automation and perturbation metrics) to evaluate the performance of the ARSENAL pipeline as we improve it. We would also like to create benchmark datasets for evaluating different aspects of ARSENAL. Apart from SAL models, we have also experimented with other logical model outputs, e.g., first-order logic. We plan to continue generating other logical models, which could be suitable for other types of formal analysis. We would also like to explore the creation of richer system models, by composing models generated from separate requirements corpora. The plug-and-play ARSENAL architecture will also facilitate trying out NL parsers other than STDP in the future.

The current ARSENAL system also has a statistics generator, which generates statistics about the distribution of entities, typed dependencies, etc. in a requirements corpus. We use the generator to identify important type rules (e.g., from dominant TDs) and important preprocessing rules (e.g., from dominant entities) for ARSENAL. We would like to use these statistics and apply machine learning to automatically customize different parts of ARSENAL (e.g., type rules, translation rules) for a given domain and requirements corpus.

In the future, we want to test ARSENAL on other domains and datasets, generate other logical models for other types of analysis (e.g., Markov Logic Networks for probabilistic analysis), and go beyond NL text to handle flow-charts, diagrams and unstructured tables in requirements. We also want to learn the type rules. There is a lot of related work on learning semantic parsers from limited user feedback, e.g., techniques for unsupervised learning [Poo12] and supervised learning from ambiguous supervision [KM07]. We would like to use a learning algorithm like FOIL [Qui90] to learn type rules — given a list of <STDP output, DSL Formula> pairs for a domain specific language, we can use FOIL-type rule learning techniques to learn the type rules.

## 8 Acknowledgments

Special thanks to Kathleen Fisher, Bruno Dutertre, Sam Owre, John Rushby, Ashish Tiwari, and Brendan Hall for all their help and guidance regarding this work. This material is based upon work supported by the United States (US) Air Force and the Defense Advanced Research Projects Agency (DARPA) under Contract Numbers FA8750-12-C-0339 and FA8750-12-C-0284. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US Air Force and DARPA.

## References

- Bab07. J. Babcock. Good requirements are more than just accurate. Practical Analyst: Practical Insight for Business Analysts and Project Professionals, December 2007.
- BCC<sup>+</sup>04. K. Barker, V. Chaudhri, S.Y. Chaw, P.E. Clark, J. Fan, D. Israel, S. Mishra, B. Porter, P. Romero, D. Tecuci, and P. Yeh. A question answering system for AP chemistry: Assessing KR technologies. In *Proceedings of International Conference on Knowledge Representation and Reasoning*, 2004.
- BCG<sup>+</sup>10. Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Konighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. Ratsy: A new requirements analysis tool with synthesis. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 425–429. Springer Berlin Heidelberg, 2010.
- BDMN06. Adam Barth, Anupam Datta, John C. Mitchell, and Helen Nissenbaum. Privacy and contextual integrity: Framework and applications. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 184–198, 2006.
- BGJ<sup>+</sup>07. R. Bloem, S. Galler, B. Jobstmann, N. Piterman, Amir Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: A case study. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, 2007.
- BGL<sup>+</sup>00. Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, jun 2000. NASA Langley Research Center.
- BKM10. David Basin, Felix Klaedtke, and Samuel Müller. Monitoring security policies with metric first-order temporal logic. In *Proceedings of the 15th ACM Symposium on Access control models and technologies*, pages 23–34, 2010.
- Bor96. J. Borstler. User-centered requirements engineering in record - an overview. In *Proceedings of Nordic Workshop on Programming Environment Research (NWPER)*, 1996.
- Boy99. N. Boyd. Using natural language in software development. *Journal of Object Oriented Programming*, 11(9), 1999.
- BP98. Barry W. Boehm and Philip N. Papaccio. Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, 14(10):1462–1477, October 1998.
- BS96. Tomas Boman and Katarina Sigerud. Requirements elicitation and documentation using T-red. Master’s thesis, Universty of Umeå, 1996.
- bug10. Software defects - do late bugs really cost more? Slashdot Article (<http://tinyurl.com/8vew93k>), March 2010.
- CFR12. M. Connor, C. Fisher, and D. Roth. Starting from scratch in semantic role labeling: Early indirect supervision. Springer, 11 2012.
- CGP99. Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.
- CK13. Shu Cai and Kevin Knight. Smatch: an evaluation metric for semantic feature structures. In *ACL*, pages 748–752, 2013.
- CM11. David L. Chen and Raymond J. Mooney. Learning to interpret natural language navigation instructions from observations. pages 859–865, 2011.
- CP10. Eric Crestan and Patrick Pantel. Web-scale knowledge extraction from semi-structured tables. In *Proceedings of International Conference on World Wide Web (WWW)*, 2010.
- cuc. Cucumber. <http://cukes.info>.
- CV96. Judith Crow and Ben L. Di Vit. Formalizing space shuttle software requirements. In *ACM SIGSOFT Workshop on Formal Methods in Software Practice*, 1996.
- dBCK<sup>+</sup>11. Berry de Bruijn, Colin Cherry, Svetlana Kiritchenko, Joel D. Martin, and Xiaodan Zhu. Machine-learned solutions for three stages of clinical information extraction: The state of the art at i2b2 2010. *JAMIA*, 2011.
- DDG<sup>+</sup>12. Rolf Drechsler, Melanie Diepenbeck, Daniel Große, Ulrich Kühne, Hoang M. Le, Julia Seiter, Mathias Soeken, and Robert Wille. Completeness-driven development. In *International Conference on Graph Transformation*, 2012.
- DJP11. Z. Ding, M. Jiang, and J. Palsberg. From textual use cases to service component models. In *Proceedings of 3rd International Workshop on Principles of Engineering Service-Oriented Systems*, pages 8–14, 2011.

- dMMM06. Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. Generating typed dependency parses from phrase structure parses. In *In Proc. Intl. Conf. on language resources and evaluation (LREC)*, pages 449–454, 2006.
- DSCS10. Dipanjan Das, Nathan Schneider, Desai Chen, and Noah A. Smith. Probabilistic frame-semantic parsing. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, HLT '10, pages 948–956, 2010.
- FFGL01. F. Fabbrini, M. Fusani, S. Gnesi, and G. Lami. An automatic quality evaluation for natural language requirements. In *Proceedings of International Workshop on RE: Foundation for Software Quality*, 2001.
- GLM08. Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2008.
- Gog96. Joseph A. Goguen. Formality and informality in requirements engineering. In *Proceedings of International Conference on Requirements Engineering*, 1996.
- GZ05. Vincenzo Gervasi and Didar Zowghi. Reasoning about inconsistencies in natural language requirements. *ACM Trans. Softw. Eng. Methodol.*, 14, July 2005.
- Har12. Ian G. Harris. Extracting design information from natural language specifications. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1256–1257, 2012.
- HS10. Michael Heilman and Noah A. Smith. Extracting simplified statements for factual question generation. In *Proceedings of AIED Workshop on Question Generation*, 2010.
- iee94. IEEE recommended practice for software requirements specifications. *IEEE Std 830-1993*, 1994.
- KGFP08. Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. Translating structured English to robot controllers. *Advanced Robotics*, pages 1343–1359, 2008.
- KM07. Rohit J Kate and Raymond J Mooney. Learning language semantics from ambiguous supervision. In *AAAI*, volume 7, pages 895–900, 2007.
- KP03. Paul Kingsbury and Martha Palmer. Propbank: The next level of treebank. In *Proceedings of Treebanks and Lexical Theories*, 2003.
- Kup12. Orna Kupferman. Recent challenges and ideas in temporal synthesis. In *SOFSEM 2012: Theory and Practice of Computer Science*, volume 7147 of *Lecture Notes in Computer Science*, pages 88–98. Springer Berlin Heidelberg, 2012.
- LDS11. Wenchao Li, L. Dworkin, and S.A. Seshia. Mining assumptions for synthesis. In *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 43–50, 2011.
- LJK11. Percy Liang, Michael I. Jordan, and Dan Klein. Learning dependency-based compositional semantics. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1*, pages 590–599, 2011.
- Mal09. Jane T. Malin. Automated tool and method for system safety analysis: 2009 progress report. Technical Report NASA/TM-2010-214800, NASA, 2009.
- Mil95. George A. Miller. Wordnet: A lexical database for English. *Communications of the ACM*, 38:39–41, 1995.
- NB09. A.P. Nikora and G. Balcom. Automated identification of LTL patterns in natural language requirements. In *20th International Symposium on Software Reliability Engineering (ISSRE)*, 2009.
- OKH07. Olga Ormandjieva, Leila Kosseim, and Ishrar Hussain. Toward a text classification system for the quality assessment of software requirements written in natural language. In *European Conference on Software Quality Assurance*, 2007.
- ORR<sup>+</sup>96. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.
- PH12. Amalinda Post and Jochen Hoenicke. Formalization and analysis of real-time requirements: A feasibility study at BOSCH. In *VSTTE*, pages 225–240, 2012.
- Poo12. Hoifung Poon. Unsupervised semantic parsing. In *2012 Symposium on Machine Learning in Speech and Language Processing (MLSLP)*, 2012.
- PP06. Nir Piterman and Amir Pnueli. Synthesis of reactive(1) designs. In *In Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 364–380, 2006.
- Qui90. J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.
- rsp. Rspec. <http://en.wikipedia.org/wiki/RSpec>.
- SACO02. Rachel L. Smith, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil. PROPEL: An approach supporting property elucidation. In *24th International Conference on Software Engineering*, 2002.
- SCK04. William Scott, Stephen Cook, and Joseph Kasser. Development and application of context-free grammar for requirements. In *System Engineering Test and Evaluation Conference (SETE)*, 2004.

- SD10. W. Steiner and B. Dutertre. SMT-based formal verification of a TTEthernet synchronization function. In *FMICS*, 2010.
- SF96. Rolf Schwitter and Norbert E. Fuchs. Attempto controlled English (ACE) a seemingly informal bridgehead in formal territory. In *JICSLP*, 1996.
- Shi02. Kanna Shimizu. *Writing, Verifying, and Exploiting Formal Specifications for Hardware Designs*. PhD thesis, Department of Electrical Engineering, Stanford University, August 2002.
- SNM10. Mihai Surdeanu, Ramesh Nallapati, and Christopher Manning. Legal claim identification: Information extraction with hierarchically labeled data. In *Semantic Processing of Legal Texts Workshop*, 2010.
- SWD12. Mathias Soeken, Robert Wille, and Rolf Drechsler. Assisted behavior driven development using natural language processing. In *Objects, Models, Components, Patterns*, volume 7304 of *Lecture Notes in Computer Science*, pages 269–287. 2012.
- TM02. S. Teufel and M. Moens. Summarizing scientific articles: Experiments with relevance and rhetorical status. *Computational Linguistics*, 28(4), 2002.
- WHH<sup>+</sup>13. Matthew R. Walter, Sachithra Hemachandra, Bianca Homberg, Stefanie Tellex, and Seth J. Teller. Learning semantic maps from natural language descriptions. In *Robotics: Science and Systems*, 2013.
- XPTX12. Xusheng Xiao, Amit M. Paradkar, Suresh Thummalapenta, and Tao Xie. Automated extraction of security policies from natural-language software documents. In *SIGSOFT FSE*, page 12, 2012.
- ZGM01. Didar Zowghi, Vincenzo Gervasi, and Andrew McRae. Using default reasoning to discover inconsistencies in natural language requirements. In *Asia-Pacific Software Engineering Conference (APSEC)*, 2001.